A DYNAMIC MULTI-THREADED QUEUING MECHANISM FOR REDUCING THE INTER-PROCESS COMMUNICATION LATENCY ON MULTI-CORE CHIPS

by

ROHITSHANKAR VIJAY SHANKAR MISHRA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Master of Science at
The University of Texas at Arlington
August, 2019

Arlington, Texas

Supervising Committee:

Dr. Ishfaq Ahmad, Supervising Professor
Dr. Ming Li
Dr. Shirin Nilizadeh

ABSTRACT

Reducing latency in Inter-Process Communication (IPC) is one of the key challenges in multi- threaded applications in multi-core environments. High latencies can have serious impact on the performance of an application when many threads queue up for memory access. Often lower latencies are achieved by using lock-free algorithms that keep threads spinning but incur high CPU usage as a result. Blocking synchronization primitives such as mutual exclusion locks or semaphores achieve resource efficiency but yield lower performance. In this paper, we take a different approach of combining a lock-free algorithm with resource efficiency of blocking synchronization primitives. We propose a queueing scheme named eLCRQ that uses the lightweight Linux Futex system call to construct a block-when- necessary layer on top of the popular lock-free LCRQ. Owing to the block-when-necessary feature, eLCRQ produces close to lock-free performance when under contention. Under no contention, we use the Futex System call for conditional blocking instead of spinning in a retry loop, which releases the CPU to perform other tasks. When compared with existing IPC mechanisms, eLCRQ yields 2.3 times reduction in CPU usage while lowering the average message latency 1.7 times. When comparing the proposed scheme with industry standard non-blocking lock-free DPDK RTE_RING, the results show a 3.4 times reduction in CPU Usage while maintaining comparable message latency. We also propose a fixed-spinning based variation of the proposed scheme, called eLCRQ-spin, which allows us to make tradeoffs between CPU usage efficiency and message latency.

CONTENTS

A Dynamic Multi-Threaded Queuing Mechnism for Reducing the Inter-Process Communication Latency on Multi-Core Chips

Rohitshankar Mishra
Department of Computer Science
University of Texas, Arlington
Arlington, Texas
rohitshankarvij.mishra@mavs.uta.edu

Dr. Ishfaq Ahmad
Department of Computer Science
University of Texas, Arlington
Arlington, Texas
ishfaq.ahmad@uta.edu

***Abstract*—Reducing latency in Inter-Process Communication (IPC) is one of the key challenges in multi-threaded applications in multi-core environments. High latencies can have serious impact on the performance of an application when many threads queue up for memory access. Often lower latencies are achieved by using lock-free algorithms that keep threads spinning but incur high CPU usage as a result. Blocking synchronization primitives such as mutual exclusion locks or semaphores achieve resource efficiency but yield lower performance. In this paper, we take a different approach of combining a lock-free algorithm with resource efficiency of blocking synchronization primitives. We propose a queueing scheme named eLCRQ that uses the lightweight Linux Futex system call to construct a block-when-necessary layer on top of the popular lock-free LCRQ. Owing to the block-when-necessary feature, eLCRQ produces close to lock-free performance when under contention. Under no contention, we use the Futex System call for conditional blocking instead of spinning in a retry loop, which releases the CPU to perform other tasks. When compared with existing IPC mechanisms, eLCRQ yields 2.3 times reduction in CPU usage while lowering the average message latency 1.7 times. When comparing the proposed scheme with industry standard non-blocking lock-free DPDK RTE_RING, the results show a 3.4 times reduction in CPU Usage while maintaining comparable message latency. We also propose a fixed-spinning based variation of the proposed scheme, called eLCRQ-spin, which allows us to make tradeoffs between CPU usage efficiency and message latency.**

Keywords— multi-producer multi-consumer, Futex, multi-threading, IPC

I. INTRODUCTION

Reducing latency in Inter-Process/Inter-Thread Communication is one of the key challenges in parallel and distributed computing. This is because as the number of threads in an application goes on increasing, usually so does the communication overhead. Hence, reducing the communication latency can have a big impact on multi-threaded application performance in multi-core environments. The mechanisms used for such communication in a Linux based OS can generally be divided into two categories – Message Passing and Shared Memory[21]. Message Passing refers to direct core-to-core communication between threads using mechanisms like Pipes, Sockets, Message Queues, Signals, etc. Shared Memory is communication via reading/writing to a shared memory segment. Some examples include POSIX Memory Mapped files and System V shared segments. All changes made to this segment will be visible to everyone else connected to the segment.

However, this advantage of immediate communication from shared memory comes with a major loophole. There is no coordination present by default, between processes writing to the shared segment. There is nothing stopping two or more separate processes/threads to write in the same memory segment at the same memory location simultaneously. This can result in unpredictable outcomes for tasks or even memory corruption in some cases.

One way to fix this is to treat the shared memory access like a critical section and guard it, using a mutual exclusion lock like Mutex/Semaphore. This way only one process/thread can access the segment at a time which will avoid memory corruption. However, as the number of threads and cores increases, this approach proves to be a bottleneck to performance. It mostly negates the performance benefit obtained by using shared memory and is usually considered very slow compared to Message Passing techniques like Pipes and Sockets. Despite being slow, locks can block threads when they cannot proceed which is very useful when CPU resources are limited.

Like the previously mentioned many types of locks, there exist many other synchronization primitives that are not implemented as part of the popular multi-threading libraries in Linux. Eventcounts is one such primitive. It consists of a Producer and Consumer Threads. When a Producer produces, it sends out a signal each time. Consumer threads listen for these signals and block if no signal has occurred before consuming. There is no one standard implementation for Eventcounts and it can be constructed using a variety of other synchronization primitives like mutex, semaphores, etc. One such primitive is the Futex Linux System call which is a contraction of Fast User level Locking. It is used to construct other heavier weight primitives Mutex and condition variables in the POSIX library. This System call can selectively put threads to sleep and wake them up as needed.

We consider a scenario, where inter-process/inter-thread communication is performed via messages being enqueued by enqueuer threads and dequeued by dequeuer threads on an underlying queuing mechanism. Decades of research has been conducted on developing Multi-producer Multi-consumer (MPMC) queues that have fast and predictable performance guarantees. These data-structures can harness the power of multi-core systems and can deliver close-to-minimum possible message latency. Lock-free MPMC

queues are implemented using compare-and-swap (CAS) that is used to synchronize between multiple threads. The result of a CAS instruction is a Boolean value indicating whether the thread which executed it was successful or not. The CAS instruction guarantees that at least one of the participating threads will succeed while all others will fail. The failed threads can simply retry for a chance to succeed and make progress. Here making progress is akin to gaining access to a resource or simply completing an operation.

However, under heavy contention, most CAS operation will fail, causing a dramatic drop in performance. This is dubbed as the CAS retry problem. For this reason, the performance of CAS based queues does not scale beyond a modest number of threads. Another atomic instruction known as Fetch-and-Add (FAA) is becoming popular for designing concurrent data structures. Unlike the CAS instruction, the FAA instruction is guaranteed to always succeed.

A common theme in lock-free algorithms is to retry till successful. Under certain conditions, lock-free algorithms can get stuck in retry loops which are far less efficient with CPU resources than locks that just puts a blocked thread to sleep. If a lot of threads are stuck in this retry loop, most of the CPU power is being wasted in order to make little progress.

We can choose to guard the resources under a mutex/condition-variable. But that defeats the purpose of using lock-free data-structures because of the negative performance impact of mutexes. But in order to use CPU resources more efficiently, threads stuck in retry loops must be put to sleep.

Our Contribution We present eLCRQ, a lock-free block-when-necessary multi-producer multi-consumer (MPMC) FIFO queue which combines lock-free performance with blocking resource efficiency. It is designed for low load scenarios in mind where the queue can randomly and frequently become empty during runtime. It features improved efficiency while under low load with a 2.3X decrease in CPU Usage and 1.7X decrease in latency compared to existing Message Passing mechanisms Pipes and Sockets on multi-core Linux based systems. It also provides a 3.4X decrease in CPU Usage while maintaining comparable latency when compared to other (MPMC) lock-free queues in low load scenarios.

II. Overview of concurrent programming

Concurrent data-structures have been a hot topic of research for decades. They offer fast predictable performance guarantees and are critical for harnessing the power of modern multi-core processors due to their affinity to multi-threaded programming. Based on the progress guarantees a concurrent data-structure provides, it can broadly be divided into three groups –

A. Obstruction free:

Using simple locking primitives like mutex/semaphores for concurrency will classify a data-structure into this category. A participating thread is guaranteed to complete an arbitrary operation, otherwise known as a critical section, in a finite number of steps when it executes in isolation. Only one thread can execute its critical section at a time while all other threads remain in blocked state. Hence, obstruction free concurrency is considered the slowest form of concurrency. This form is also susceptible to many known concurrency problems like deadlocks, live-locks, race conditions, etc. However, in most practical scenarios, obstruction free concurrency is chosen because it is much simpler to reason about, is very resource efficient, and can be finely tuned to match the performance of lock free and wait free approaches.

B. Lock free:

Using atomic instructions like CAS and FAA, lock free techniques allows multiple threads to work together in a non-blocking way. As the name implies, locks are not used in their implementation. Some participating thread is guaranteed to complete an arbitrary operation, otherwise known as critical section, in a finite number of steps. Even if progress is made by some thread, it is possible for a random thread to be starved of CPU time. The progress is made in a non-blocking way. Hence, other threads are free to retry or perform some other task. Unlike obstruction free algorithms, lock free algorithms are resource hogs and may bottleneck the performance of a program if not used correctly.

C. Wait free:

Wait free algorithms, like their lock free counterparts, are similarly based on atomic instructions like CAS and FAA and are also non-blocking in nature. But they offer a stricter progress guarantee than lock free algorithms. While lock free guarantees that some random thread will complete its critical section in a finite number of steps, wait free guarantees that all threads will complete their critical sections in a finite number of steps at the same time. This restriction rules out the possibility of starvation for all threads. However, practical wait free algorithms are very hard to design and are usually considered inefficient compared to their lock free equivalents.

III. Overview of IPC Techniques in Linux

First, confirm that you have the correct template for your paper size. This template has been tailored for output on the A4 paper size. If you are using US letter-sized paper, please close this file and download the Microsoft Word, Letter file.

A. PIPES:

Pipes is the simplest form of IPC. It is implemented on every flavor of Linux and implements basic one-way communication. Anything can be written to the pipe, and read from the other end in the order it came in. By default, pipes are blocking in nature and the read end will always block until data arrives to be read. This means that the reader will not waste CPU cycles if the pipe is empty. Pipes are usually the simplest and the fastest way to implement IPC in Linux environments. There also exists another variant of pipe called a FIFO or named pipe. The difference is that a FIFO has a name attached to it and can be opened by unrelated processes by referencing its name.

B. Sockets:

Sockets provide two-way communication over a wide variety of domains. While sockets are generally used to communicate over the Internet domain, we will be discussing them in the context of the Unix domain. The difference between the two is that Unix domain sockets communicate

entirely within the operating system kernel. The different types of sockets are –

1) Stream Socket: These provide sequenced, reliable, and unduplicated flow of data. This type of socket can read an arbitrary number of bytes, while still preserving the byte sequence. A stream operates much like a telephone conversation.

2) Datagram Socket: A datagram is unit of data sent as a packet. A datagram socket chunks the information to be sent in the form of datagrams. The destination socket may receive messages in a different order from the sequence in which the messages were sent. Each packet of data is individually addressed and routed. Datagram sockets operate much like passing letters back and forth in the mail.

3) Sequential Socket: This type of socket is a combination of the Steam socket and Datagram socket. It sends messages in datagram packets like the Datagram socket while at the same time ensuring sequential consistency of those packets like the Stream socket.

C. Message Queues:

Both the System V and POSIX libraries offer this form of IPC mechanism. It allows processes to exchange data in form of variable length messages. The messages can also have different message priorities or tokens associated with them. Messages are stored in the queue in order of their priorities. This allows a receiver to narrow its scope and only receive messages of a specific priority. It can also be used to identify the sender in case of multiple senders. However, there is a limit on the number of message queues and the size of each message which is defined by the operating system.

D. Semaphores:

Semaphores are one of the most popular signaling mechanisms and like message queues, they are implemented by both System V and POSIX libraries. Unlike the previous types of IPC, semaphores cannot send or receive messages. Instead, they are used to control accesses to critical sections. Here, critical sections can be anything including but not limited to file, shared memory, or network operation. The semaphore itself is represented by a non-zero integer value. This integer value can then either be incremented or decremented. If the value of a semaphore is already zero, the decrement operation will block until the value is incremented by some other process/thread.

E. Mutex:

Like Semaphore, a Mutex (short for mutual exclusion) is another synchronization primitive that is used to coordinate access to critical sections. But unlike semaphores, it is a locking mechanism instead of a signaling mechanism. A simple mutex object has two states – locked/unlocked. A thread must lock a mutex object to proceed into its critical section. Any other thread which tries to lock an already locked mutex will be blocked and added to a waiting list. Once the original thread unlocks the mutex, it will be locked again by the previously blocked thread which can then proceed into its own critical section. This is how mutual exclusion is enforced by a mutex.

F. Condition Variables

Condition variables are signaling mechanisms which are always used in conjunction with Mutexes. While a Mutex can synchronize thread accesses to data, condition variables allow threads to synchronize based on the actual value of data. They are used in scenarios when a thread needs some condition to be true before entering its critical section. Without this primitive, a thread would acquire a mutex lock and continuously poll a data value to check if a condition is true. This can be very resource intensive and inefficient especially since there might be other threads waiting on the Mutex lock. With the use of condition variables, the thread can release the Mutex lock for other threads to use while it is waiting for a signal that the condition is true. Once it receives a signal, it can the reacquire the Mutex lock and proceed unobstructed into its critical section.

G. Barriers

In multi-threaded programming, many times a scenario arises where all threads must complete their assigned task before any of them can move onto the next task. The Barrier synchronization primitive has been designed exactly for such scenarios. In the POSIX library, the number of threads that need synchronization is specified during initialization of the barrier. At runtime, when a thread reaches a barrier synchronization point in code, it gets blocked till the required number of threads reach the same point. Hence, lagging threads get a chance to catch up to other threads. This effectively ensures that all the threads will begin the next task at the same time.

H. Futex

Futex is short for *Fast User-level locking*. As the name suggests, it is a fast, lightweight kernel-assisted locking primitive. They are the building blocks upon which POSIX Mutexes, condition variables, Semaphores, rwlocks, barriers, eventcounts, etc. are built. The futex state is stored in a user-space 32 bit variable. This state is manipulated using atomic operations and in the uncontended case without the overhead of a kernel system call. If the lock is contended, the kernel is involved to block and wake up threads. A thread can request to be put in a blocked state by passing an expected value to the Futex system call as a parameter. If the current value of the 32-bit Futex state matches the expected value parameter, the requesting thread is put in blocked state. This operation is analogous to an atomic compare-and-block operation. Similarly, to wake up threads waiting on a Futex, another Futex System call can be made with the number of threads to be woken up sent as a parameter. Usually, this number is set to 1 or INT_MAX (to wake up all waiting threads).

I. RTE_RING:

The Data Plane Development Kit (DPDK) is a collection of high-performance libraries to accelerate packet processing workloads running on a wide variety of CPU architectures. The Ring library is one such library in DPDK. It consists of a Ring Manager which is a fixed-size non-blocking queue, implemented as a table of pointers. It features FIFO access, lock-free implementation, Multi- or single-consumer enqueue/dequeue, and bulk enqueue/dequeue. The lock-free implementation is purely CAS based. The main features of DPDK libraries is that instead of requesting memory allocation from the OS, it uses Linux hugepages. Because of

this, it can transfer large amounts of data with comparatively low overhead.

IV. Related Work

We refer the reader to the detailed survey conducted in Michal Scott's paper[5] and Steven Smith's[20] paper for additional work that predates theirs.

A. MSQueue

Michael and Scott presented one of the first CAS based non-blocking queues (MSQueue)[5]. More information on related atomic instructions is provided later in this paper. This queue used a simple linked list as its base and maintained head and tail pointers at the enqueue and dequeue heads respectively. These head and tail pointer are always modified using the atomic CAS instruction. At the time, this algorithm was the fastest and most practical non-blocking queue. Many attempts have been made [11,12,13] to improve upon this implementation, but all of them suffer from one major flaw. At higher levels of concurrency, the CAS atomic instruction will succeed only for one thread at a time. Most of the threads will be stuck in a CAS retry loop which leads to a fall of in performance beyond a modest number of threads. Hence, the original MSQueue and most of the queues which were based on this idea suffer from scalability problems.

B. LCRQ

Unlike MSQueue which uses linked list as its base data structure, LCRQ uses Concurrent Ring Queues (CRQs) as their base. CRQs are cyclic arrays that use a single fixed-size buffer as if it was connected end-to-end. The most useful property of CRQs is that the beginning and end of the buffer is relative, and elements need not be shuffled around when one is removed from the array. This makes CRQs well-suited for FIFO data structures. However, the maximum size of any such CRQ cannot be changed during runtime. This restriction reduces the flexibility on the number of elements which can be stored in any one CRQ.

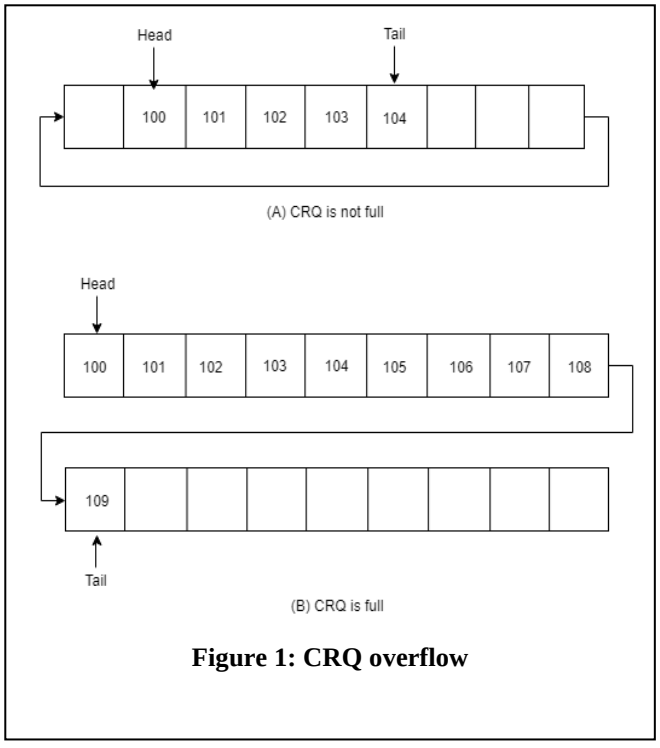

Figure 1: CRQ overflow

LCRQ (List of CRQs) is a non-blocking Multi-producer Multi-consumer FIFO queue which uses the FAA atomic instruction in conjunction with the CAS instruction to achieve high scalability. It consists of a linked list of CRQs as its base. This overcomes the problem of a fixed size CRQ and makes LCRQ effectively unbounded. In addition, it uses the FAA atomic instruction to spread enqueuers and dequeuers around while minimizes contention. This helps LCRQ to avoid the CAS based scalability problems faced by prior queue implementations like MSQueue. Initially, LCRQ only consists on one CRQ. When this fills up, a new CRQ is allocated and attached to the next pointer of the initial CRQ. All enqueuers are moved to the new CRQ while dequeuers work on emptying the old CRQ. When the old CRQ become empty, it is deallocated from the memory and dequeuers move onto the next CRQ. At the time of writing this paper, LCRQ is currently the fastest lock-free queue in literature. Its benchmarks outperform previous implementations by up to 2.5X on multi-core processors. For this reason, it is chosen as the base of the FIFO queue described in this paper.

However, we focus on improving the resource efficiency of this queue while making suitable tradeoffs in scalability and performance.

C. Futex based primitives

Fast User space muTEX, Futex for short, is a locking primitive that is a base for many other synchronization primitives including but not limited to Mutexes, condition variables, Semaphores, and barriers, in the Linux POSIX library. The Futex is implemented as a System call in the Linux operating system but has no explicit glibc wrappers. In [14], the Futex system call is used to implement several atomic primitives with encouraging results.

The Eventcount mechanism was first described by Vyukov D. as a proposal for use in Intel's Thread Building Blocks (TBB) library. The original implementation made use of Mutexes/Semaphores to support multiple Operating Systems. However, this mechanism can also be implemented using the previously mentioned, Linux system call Futex for further improving its efficiency[22]. A detailed description of the working of Eventcount is included later in this paper.

V. The Proposed Mechanism

We now present the eLCRQ algorithm with LCRQ as a black box. eLCRQ is a combination of a Futex based Eventcounts mechanism based on top of a LCRQ.

Eventcount can best be described as a condition variable for lock-free algorithms. The problem with traditional mutex/condition variables and semaphores is that invoking them will usually result in a call to the kernel. The kernel will then decide to block/unblock the thread based on the current state. This incurs significant overhead in terms of latency. Eventcount is a synchronization primitive that facilitates user space handling of such scenarios. The blocking/unblocking logic of Eventcounts is isolated in a separate layer from the lock-free algorithm and is executed in two phases. The first phase determines if blocking is necessary (slow path) or not (fast path). It does so by checking an arbitrary condition which is supplied by the user. If the supplied condition is already true, no blocking is necessary, and the lock-free algorithm can execute as usual.

```
1. enqueue(x: Object) {
2.   LCRQenq(x)
3.   notify() // wake blocked consumers
4.}
```

Figure 2: Eventcount enqueue

```
1 dequeue() {
2  x := LCRQdeq() // spin for eLCRQ-spin
3  if(x is null) { // handle fast path
4   while(True) { // guard spurious wakes
5    key = prepareWait() // slow path
6   x := LCRQdeq()
7    if(x is null) {
8     wait(key) // block
9    } else {
10    cancelWait()  // cancel block
11    break
12   }
13  }
14 }
15 return x
16}
```

Figure 3: Eventcount dequeue

However, if the condition is false, the thread must make necessary preparations to block until the condition becomes true.

In context of a Producer-Consumer queue, the condition which can be supplied to the Eventcount can be to check whether the queue buffer is empty. If the first check returns False, i.e., the queue is not empty, the consumer can proceed along the fast path, bypassing the Eventcount mechanism altogether. If the first check returns True, i.e., the queue is empty, the consumer must prepare to block along the slow path. Once a producer thread enqueues an element in the queue, it notifies a blocked consumer thread which unblocks and proceeds to dequeue the element. Figures 2 and 3 describe the entire scenario in pseudocode. A second version called eLCRQ-spin includes a fixed amount of spin at *Line 2* in Figure 3. This allows for tradeoffs between CPU usage and acceptable latency. LCRQenq and LCRQdeq are stand-ins for the default non-blocking enqueue and dequeue operations already present in LCRQ. A brief description has been provided later in this paper. For implementation details and benchmarks against competing queues, please see [1].

A. *LCRQenq*

Following our earlier terminology of Producer-Consumer queue, a Producer thread performs the LCRQenq operation on a CRQ as shown in Figure 1. Initially, a producer thread performs an atomic FAA operation of the tail of the current CRQ. This returns the index of the CRQ on which enqueue operation is to be performed and atomically increases the tail to the next index for use by other threads. A 64-bit CAS operation is then performed on this index to enqueue an element on the current index. If the CAS operation succeeds, then the enqueue operation completes successfully. If it fails, then the producer thread retries. The enqueue operation can fail repeatedly in two possible scenarios. Either the current CRQ is full, or the current producer is being starved by other threads. In both scenarios, the current CRQ is marked as closed by the producer and a new CRQ is created. All producers then move onto the new CRQ and the current CRQ is closed off to any new enqueue operations. Once all consumer threads finish dequeuing from the current CRQ, it is deallocated from the memory and all operations are now performed on the new CRQ.

B. *LCRQdeq*

The non-blocking dequeue operation consists of a consumer thread returning a value from an index in a CRQ and replacing that index with an EMPTY value. The consumer thread performs an atomic FAA operation to get the head index of the current CRQ. Since, this operation is atomic, other consumer threads are advanced to further indexes. Then, an atomic read is performed to retrieve the current index value of the node. If node index > head index, then the dequeue operation has been overtaken between the FAA and the atomic read. In this case, the CRQ might be empty which just returns an EMPTY value from the dequeue operation. However, if the queue is not empty, the entire operation is repeated by again performing an FAA on the head. If the value of the node index is consistent with the value of the head index after the atomic read, a CAS operation is performed to replace the current node value with an EMPTY and return the current value. If this operation fails or the current node value is EMPTY, the consumer thread moves onto the next index to retry another dequeue operation. Between each retry, a check is made to see if the queue is empty. If it is, the dequeue simply returns an EMPTY value instead of retrying.

VI. Experimentation and Evaluation

A. *Evaluated Algorithms*

We compare eLCRQ and eLCRQ-spin to the best performing IPC techniques on a Linux system, namely Pipes, and Stream sockets. We also test performance against industry standard Intel DPDK's RTE_RING, the classic non-blocking MSQueue, and base LCRQ.

From [20], we know that Linux Pipes and Sockets are one of the fastest and simplest forms of Inter-Process Communication. They also adapt well to multi-producer multi-consumer scenario. Pipes are implemented by utilizing the *pipefs* virtual file system which is maintained in the memory and the *pipe* system call. The maximum size of a message that can be sent through a pipe is 64 KB. Similarly, the socket is uses the *sockfs* and the *socket* system call to create and maintain a socket. In case of a stream socket, a message is treated as a continuous stream of bytes.

RTE_RING manages memory on its own instead of relying on the operating system and kernel. It does this by accessing Linux Hugepages, which is a portion of RAM, typically with sizes ranging from 4 KB to 2 GB. By making use of Hugepages, the kernel has fewer pages in memory to deal with which mean less overall overhead in

accessing/maintaining them. This library also makes use of a fixed-size CAS based ring array as the base data-structure which further increases performance.

B. Implementation

We implement simulations related to Pipes, Sockets, MSQueue, LCRQ, eLCRQ, and eLCRQ-spin in C++, except RTE_RING which is a C only library. The algorithms for MSQueue and LCRQ where referenced from [24] which use

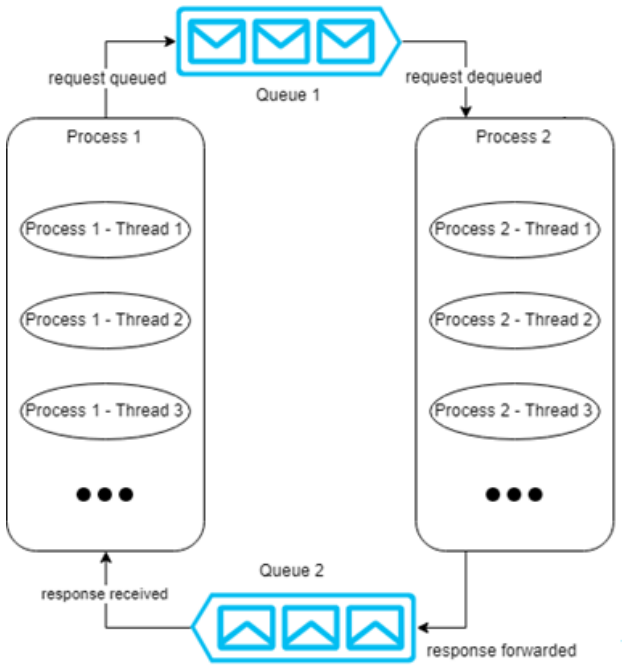

Figure 4: Simulation

Hazard Pointers[10] for garbage collection. Synchronization barriers are placed at the start for every IPC mechanism to exclude the time taken for initialization. This also ensures all producer and consumer threads start at the same time, thus eliminating any inconsistencies in timing latency between sending/receiving.

C. Platform

We use the CentOS 7 Linux distro running on an Intel Core i9-7920X 12-core processor for running our benchmarks. Each core has a base clock of 2.90 GHz with Turbo Boost up to 4.3 GHz which multiplexes into two hardware threads. This brings the total system core count to 24 cores. As is usual for Intel processors, each core has private L1 and L2 caches with shared L3 cache. Memory frequency is clocked at 3 GHz.

D. Methodology

The evaluation consists of a ping-pong IPC test scenario. It is made up of two separate processes each with its own set of producer and consumer threads. Producer threads from Process 1 communicate with Consumer threads in Process 2 and vice versa. The underlying communication mechanism between the two processes is changed between Pipes, sockets, MSQueue, LCRQ, RTE_RING, eLCRQ, and eCLRQ-spin. Figure 4 is a pictorial representation of the simulation. Two different benchmarks are performed for each mechanism. A latency benchmark which records the time delay between messages being sent and received between producer-consumer pairs, and a CPU usage benchmark which tracks the average CPU usage during the simulation. To provide a result that is relatively free from noise due to background tasks, we run every benchmark 5 times with 1 million iterations each. This helps the system to reach a steady state during each benchmark. The simulation is repeated for varying number of threads ranging from 2 to 24. This is done to study the scalability of every mechanism. In addition to this, each thread is manually pinned to distinct cores to eliminate variances due to random nature of the kernel task scheduler.

E. Results

Figures 5 and 6 show the results of benchmarks with different IPC mechanisms. Examining the results from the two benchmarks, provides significant insight at the efficiency of existing IPC mechanisms in the Linux kernel. We omit the results of the Datagram socket and Sequential socket because we found them to perform very close to the Stream socket. We will discuss results in two parts, considering the *Latency* and *CPU Usage* respectively.

1) Latency:

Here we measure the time difference in sending and receiving a 64-bit integer between producers and consumers.

From Figure 5, we see that eLCRQ improves upon existing message passing Linux IPC mechanisms Pipes and Sockets by 1.7X at 24 threads. This is to be expected because Shared memory communication is the fastest form of IPC communication. Because the eCLRQ Eventcount mechanism rarely goes into slow path during this benchmark, the use of atomic instructions for synchronization increases the likelihood of higher throughput

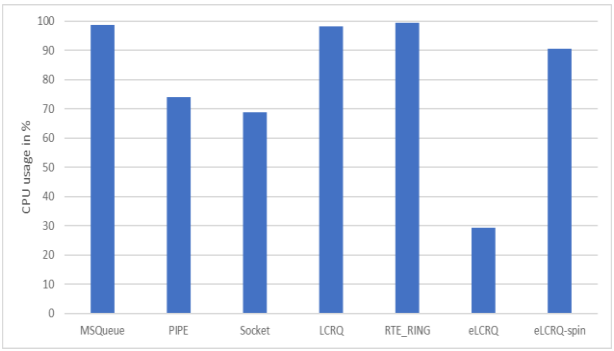

Figure 6: CPU usage benchmark (lower is better)

Purely CAS instruction based MSQueue suffers quite a bit at higher concurrency levels due to the CAS retry problem. This leads to the worst scaling queue for IPC by a significant margin. eLCRQ outperforms MSQueue by 1.9X at 24 threads.

RTE_RING performs much better than MSQueue even though it uses the CAS instruction for manipulating its queue. This is because it avoids the memory management overhead of the Linux kernel by using Linux Hugepages and managing memory on its own. It manages to beat the eLCRQ by a margin of 1.3X but loses to eLCRQ-spin by 5.61X.

As expected, the original LCRQ beats all other queues because of its efficient use of the FAA atomic instruction to

spread multi-threaded load in addition to the CAS instruction. It is the fastest non-blocking queue in literature at the time of writing of this paper. However, eLCRQ-spin very closely matches the performance of the original LCRQ.

2) CPU Usage

CPU Usage is an important metric when considering the use of any algorithm in a practical important. It provides important hints into its implementation and scalability. To test this metric in a realistic scenario where each IPC mechanism is subjected to random delay in messages, a small delay of 1 – 20 us in introduced after each producer-enqueue/consumer-dequeue. Only the CPU Usage of Consumer threads is measured in this simulation.

From Figure 6, blocking Pipes and Sockets have good CPU Usage characteristics when compared to other non-blocking mechanisms. eLCRQ however improves upon them by 2.3 X at 24 threads.

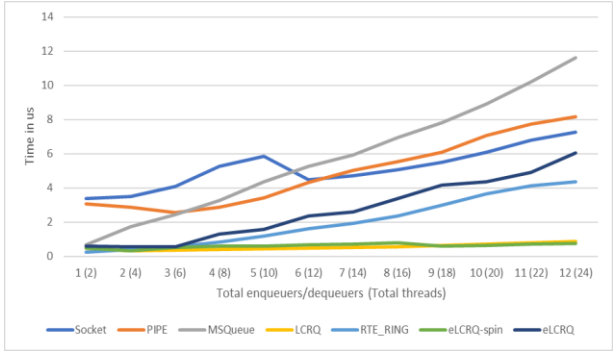

Figure 5: Scaling test (lower is better)

As expected by non-blocking mechanisms, RTE_RING, and MSQueue both take close to 100 % CPU resources. eLCRQ takes up upto 3.4 X while eLCRQ-spin takes 1.1 X less CPU resource at 24 threads, during the simulation.

Overall, eLCRQ takes up less CPU resources than other competing IPC mechanisms in this simulation. This is made possible due to efficient waiting on dequeue requests by the Futex-based Eventcount layer.

VII. Conclusion

We have presented eLCRQ, a concurrent non-blocking block-when-necessary queue based on LCRQ and Futex based event-counts that outperforms standard Linux IPC mechanisms in terms of per message latency by 1.7X in its base form and by more than 9.2X with fixed spinning. It also beats standard Linux IPC and industry standard RTE_RING in terms of CPU usage on randomly delayed loads by 3.4X.

References

[1] Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. In Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '13). ACM, New York, NY, USA, 103-112.

[2] Chaoran Yang and John Mellor-Crummey. 2016. A wait-free queue as fast as fetch-and-add. In Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16). ACM, New York, NY, USA, Article 16, 13 pages.

[3] X. Meng, X. Zeng, X. Chen and X. Ye, "A cache-friendly concurrent lock-free queue for efficient inter-core communication," 2017 IEEE 9th International Conference on Communication Software and Networks (ICCSN), Guangzhou, 2017, pp. 538-542.

[4] Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing (PODC '96). ACM, New York, NY, USA

[5] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A persistent lock-free queue for non-volatile memory. In Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18). ACM, New York, NY, USA, 28-40.

[6] Gal Milman, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. 2018. BQ: A Lock-Free Queue with Batching. In Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA '18). ACM, New York, NY, USA, 99-109.

[7] Deli Zhang and Damian Dechev. 2016. A Lock-Free Priority Queue Design Based on Multi-Dimensional Linked Lists. IEEE Trans. Parallel Distrib. Syst. 27, 3 (March 2016), 613-626.

[8] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. 2012. An efficient unbounded lock-free queue for multi-core systems. In Proceedings of the 18th international conference on Parallel Processing (Euro-Par'12), Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis (Eds.). Springer-Verlag, Berlin, Heidelberg, 662-673.

[9] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. IEEE Trans. Parallel Distrib. Syst. 15, 6 (June 2004), 491-504.

[10] M. Hoffman,O. Shalev,and N.Shavit. The baskets queue. InOPODIS 2007.

[11] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free FIFO queues. In DISC 2004.

[12] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free FIFO queues. In SPAA 2005.

[13] Jens Gustedt. 2016. Futex based locks for C11's generic atomics. In Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16). ACM, New York, NY, USA, 2004-2006.

[14] Hadi Asgharimoghaddam and Nam Sung Kim. 2016. SpinWise: A Practical Energy-Efficient Synchronization Technique for CMPs. SIGARCH Comput. Archit. News 44, 1 (July 2016), 1-8.

[15] Davidlohr Bueso. 2016. Futex Scaling for Multi-core Systems. In Applicative 2016 (Applicative 2016). ACM, New York, NY, USA

[16] Jeff Bonwick. 1994. The slab allocator: an object-caching kernel memory allocator. In Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1 (USTC'94), Vol. 1. USENIX Association, Berkeley, CA, USA, 6-6.

[17] Franke, Hubertus & Russell, Rusty & Kirkwood, Matthew. (2002). Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux.

[18] Drepper, Ulrich. (2004). Futexes Are Tricky.

[19] Steven Smith, Anil Madhavapeddy, Christopher Smowton, Malte Schwarzkopf, Richard Mortier, Steven H and Robert M. Watson, “Draft: Have you checked your IPC performance lately?,”

[20] Silberschatz, A., Galvin, P. and Gagne, G. (2014). Operating system concepts. Hoboken, N.J: Wiley.

[21] Hall, B. (2019). Beej's Guide to Unix IPC. [online] Beej.us. Available at: https://beej.us/guide/bgipc/ [Accessed 22 Jul. 2019].

[22] GitHub. (2019). facebook/folly. [online] Available at: https://github.com/facebook/folly/tree/master/folly [Accessed 22 Jul. 2019].

[23] Vyukov, D. (2019). Eventcounts - 1024cores. [online] 1024cores.net. Available at: http://www.1024cores.net/home/lock-free-algorithms/eventcounts [Accessed 22 Jul. 2019].

[24] Ramalhete, P. and Correia, A. (2019). pramalhe/ConcurrencyFreaks. [online] GitHub. Available at: https://github.com/pramalhe/ConcurrencyFreaks/tree/master/CPP/queues [Accessed 22 Jul. 2019].

[25] Vyukov, D. (2019). Eventcounts - 1024cores. [online] 1024cores.net. Available at: http://www.1024cores.net/home/lock-free-algorithms/eventcounts [Accessed 22 Jul. 2019].